Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1968 – Mars 2011

On the Implementation of Concurrent Objects*

Michel Raynal**

Abstract: The implementation of objects shared by concurrent processes, with provable safety and liveness guarantees, is a fundamental issue of concurrent programming in shared memory systems. It is now largely accepted that linearizability (or atomicity) is an appropriate consistency condition for concurrent objects. On the liveness side, progress conditions (mainly absence of deadlock or the stronger absence of starvation) have been stated and investigated since a long time and are now well-mastered. The situation is different in asynchronous shared memory systems prone to process failures.

This paper visits three progress conditions suited to concurrent objects in presence of failures, namely obstruction-freedom, non-blocking and wait-freedom. To that end, the paper visits also appropriate computation models and paradigm problems to illustrate this family of progress conditions. The paper has consequently an introductory and survey flavor. Its aim is to help people better understand the difficulties, subtleties and beauties encountered when one has to implement concurrent objects despite the net effect of asynchrony and failures.

Key-words: Asynchronous shared memory system, Atomicity, Compare&Swap, Consensus object, Consensus number, Enriched system, Failure detector, Linearizability, Lock, Lock-freedom, Obstruction-freedom, Non-Blocking, Process crash, Progress condition, Queue, Read/Write atomic register, Set, Snapshot, Splitter, Synchronization, System boosting, Timestamp, Wait-free algorithm.

Sur l'implémentation des objects concurrents

Résumé : *Ce rapport s'intéresse à la mise en œuvre des objects concurrents.*

Mots clés : *Système asynchrone, mémoire partagée, objet concurrent, tolérance aux fautes.*

---

* To appear in a Springer Verlag LNCS volume dedicated to Brian Randell.

** Membre senior de l'IUF et Projet ASAP: équipe commune avec l'INRIA, le CNRS et l'université Rennes 1, raynal@irisa.fr.

Preamble This paper has been written for the *Festschrift* volume celebrating Brian Randell's 75th birthday. When I was contacted to participate to this *liber amicorum*, I immediately remembered a famous and seminal paper of Brian. This paper, titled "*Process structuring*" and co-authored with J.J. Horning, was published in 1973 [23]. I was a PhD student at that time, and the reading of this paper (together with a paper by Dijkstra [9]) illuminated my view of what a process is and what synchronization is. (I would like to encourage young people to read these papers for their clarity and the ideas they have introduced and developed 40 years ago!) Hence, when I was asked if I was interested in contributing to the *Festschrift* volume, my answer was at once "yes", and the topic was evident: recent results in fault-tolerant concurrency and synchronization.

1 Introduction

1.1 Consistency and progress conditions for concurrent objects

Concurrent objects A *concurrent object* is an object that can be concurrently accessed by several processes. As any object, a concurrent object is defined by a set of operations that processes can invoke to cooperate through this object. These operations are the only way to access the internal representation of the object (that remains otherwise invisible to processes).

We are interested here in concurrent objects that have a *sequential specification* and supply processes with *total operations*. A total operation is an operation that always returns a result (e.g., a dequeue() operation on an empty queue returns the value *empty* instead of blocking the invoking process).

Linearizability The most popular safety property associated with concurrent objects is called *linearizability* [20]. This consistency condition extends *atomicity* to all objects defined by a sequential specification on total operations. Hence, an implementation of an object satisfies *linearizability* (and we say that the object implementation is *linearizable*) if the operation invocations issued by the processes appear (from an external observer point of view) as if they have been executed sequentially, each invocation appearing as being executed instantaneously at some point of the time line between its start event and its end event. Said differently, an implementation is linearizable if it could have been produced by a sequential execution.

An important property associated with linearizable object implementations is the fact that linearizable implementations compose for free. This means that, if each of the implementations of an object A and an object B (each taken independently) are linearizable, then these implementations without any modification constitute a linearizable implementation of the composite object (A, B) (i.e., an object made up of (A and B). It is important to notice that, in contrast to linearizability, neither sequential consistency [28] nor serializability [5] are consistency conditions that compose for free.

Traditional lock-based shared memory synchronization One of the most popular way to obtain linearizable implementations of concurrent objects is to use locks. Associating a single lock with an object prevents several processes/threads from accessing it simultaneously. This approach is based on the classical notion of mutual exclusion [9, 38, 45]. Interestingly, locks can take different shapes according to the abstraction level at which they are considered. The most known example of locks is certainly the *semaphore* object [9], on top of which more friendly (i.e., high level) locks-based abstractions (such as monitors [22] or serializers [21]) can be built. This approach has proved its usefulness in providing solutions to basic paradigms of shared memory synchronization (such as the producer-consumer problem, or the readers-writers problem). One of the main difficulties when one has to design a lock-based solution lies in ensuring deadlock prevention, and more generally, provable fairness guarantees.

Limit of locks Using lock-based mutual exclusion to implement linearizable concurrent objects has two major drawbacks. First, using locks to protect large pieces of data can drastically limit concurrency and consequently reduce efficiency. Although tricky, this issue can be solved by clever programmers who use locks on small pieces of data in order to favor parallelism.

The second and more severe drawback due to locks appears when one wants to use them in failure-prone systems [40]. Let us consider an asynchronous system in which processes (a) can crash and (b) communicate only by reading and writing a shared memory. If a process locks an item and crashes before releasing the lock, no process can know if this process is very slow (e.g., due a page fault or an input/output) or has crashed. (More generally, this impossibility to distinguish between these scenarios make some problems impossible to solve in failure-prone asynchronous read/write systems.) This means that, according to the failure pattern, locks can entail the permanent blocking of processes in these systems.

Progress conditions Several progress conditions (liveness properties) for concurrent objects have been proposed for failure-prone asynchronous systems. We consider three of them here.

- Obstruction-freedom. An implementation of a concurrent object is *obstruction-free* if it guarantees that any process p that executes an operation eventually terminates if it (p) executes alone during a long enough period of time (without crashing) [18]. (In some papers, it is said that obstruction-freedom guarantees termination only in *solo* executions.)

  The wording "long enough period" is due to asynchrony. From a practical point of view, this means that, due to asynchrony, no upper bound on the time needed to execute an operation can be known. It is important to notice that obstruction-freedom does not guarantee termination if no process executes solo for long enough. This progress condition is meaningful in systems in which, while conflict can happen, they are rare (we will see in Section 6 how conflicts can be solved in these rare occasions).

- Non-blocking. An implementation of a concurrent object is *non-blocking* if it guarantees that at least one of the processes that have invoked an operation on the object will terminate its invocation whatever the concurrency pattern and the behavior of the other processes.

  It is easy to see that non-blocking is a progress condition stronger than obstruction-freedom. It can be seen as deadlock-freedom despite asynchrony and process crashes.

- Wait-freedom. An implementation of a concurrent object is *wait-free* if it guarantees that any process that has invoked an operation terminates it (if it does not crash) whatever the concurrency pattern and the behavior of the other processes [16]. This is the strongest possible progress condition.

  It is easy to see that wait-freedom is a progress condition stronger than non-blocking. It corresponds to $(n-1)$-*resilience* (where n is the total number of processes). It provides starvation-freedom despite asynchrony and any number of process crashes.

It follows from the previous definitions that obstruction-freedom, non-blocking and wait-freedom define a hierarchy of progress conditions for the implementation of concurrent objects. Moreover, it is easy to see that obstruction-free, non-blocking and wait-free implementations are necessarily lock-free (i.e., they cannot be based on locks).

1.2 Content of the paper

The paper is an introduction to lock-free, obstruction-free, non-blocking and wait-free implementations of concurrent objects. To that end, it first introduces the base asynchronous read/write computation model and associated enriched models (Section 2). Then it presents algorithms implementing objects with the previous liveness properties.

- Section 3 presents a lock-based implementation of a concurrent set object due to Heller, Herlihy, Luchangco, Moir, Scherer and Shavit [15]. Such an object has three operations: add(), remove() and contain(). As we will see, add(), remove() use locks on at most two items while the operation contain() is lock-free. This implementation is particularly efficient when the number of add() and remove() operations is small with respect to the number of contain() operations.

- Section 4 describes a non-blocking implementation of a queue due to Michael and Scott [34]. This implementation considers an asynchronous read/write system enriched with a Compare&Swap operation.
- Section 5 describes wait-free implementations of two objects suited to the base read/write shared memory model. The first object has been implicitly introduced by Lamport [30]. It has then been given an object status in [37] where it is called *splitter*. This object can be used to implement more sophisticated objects. The second object, called the *snapshot* object, is a fundamental object for fault-tolerant computing in asynchronous read/write shared memory systems [2].
- Section 6 is focused on obstruction-freedom. It first presents and obstruction-free implementation of a concurrent timestamping object. It then addresses the following question: how to boost an obstruction-free implementation to obtain a wait-free implementation? A failure detector-based answer proposed by Guerraoui, Kapalka and Kuznetsov is presented [11].

Finally, Section 7 concludes the paper with a few remarks on the notions of *consensus* and *universal construction* [16], which is a consensus-based construction that allows the design of wait-free implementations of any object defined by a sequential specification.

2 Computation model

2.1 Base read/write computation model

Process model The system consists of n sequential processes denoted p_1, p_2, ..., p_n. The integer i is called the index of p_i. The processes are asynchronous. This means that the relative execution speed of different processes is arbitrary, and there is no bound on the time it takes for a process to execute a step.

Failure model A process may crash (halt prematurely). After it has crashed, a process executes no step. A process executes correctly until it possibly crashes. A process that does not crash in a run is *correct* in that run. Otherwise it is *faulty* in that run. Let t be a model parameter that defines the upper bound on the number of processes that may crash in a run. In the following we consider the case $t = 0$ (failure-free model) and the case $t = n - 1$ (wait-free model).

Communication model In the base model, the processes communicate through multi-writer/multi-reader atomic registers. "Atomic" means the read and write operations appear as if they have been executed (1) sequentially, (2) the order being such that if an operation $op1$ terminates before an operation $op2$ starts then $op1$ appears before $op2$ (said in another way, the total order respects the real-time occurrence order on operations). The words "atomicity" and "linearizability" are synonym. The word "atomicity" is generally used for shared registers [29], while the word "linearizability" is used mainly for arbitrary objects defined by a sequential specification [20].

The registers are assumed to be reliable. This assumption is without loss of generality as it is possible to build atomic reliable registers on top of crash prone atomic registers [4, 13, 33]. Moreover, it is assumed that each atomic register can contain arbitrary values.

Notation Atomic registers and all the objects that are in the shared memory are denoted with upper case letters. Differently, all the local variable are denoted with lower case letters; the process index is sometimes used as a subscript for local variables.

In the algorithms that are presented, some atomic registers contain pointers. The following notations are used with respect to pointers. If P is a pointer, $P\downarrow$ denotes the object pointed to by P. If X is an object, $\uparrow X$ denotes a pointer that points to X. Hence, $(\uparrow X)\downarrow$ and X denote the same object.

Default value The value $\perp$ and $\top$ are default values that are used only by the algorithms. This means that they are unknown to the processes everywhere else.

2.2 Enriched computation models

Each of the algorithms that are presented is designed for a given computation model. We distinguish here the following models. The acronym $\mathcal{ASM}$ stands for $\mathcal{A}$synchronous $\mathcal{S}$hared $\mathcal{M}$emory.

System model $\mathcal{ASM}_{n,0}[\emptyset]$ This is the pure failure-free ($t = 0$) read/write asynchronous model. In this model, atomic registers are the only way for processes to communicate.

System model $\mathcal{ASM}_{n,0}$[lock] This computation model is the failure-free $\mathcal{ASM}_{n,0}[\emptyset]$ model enriched with locks. Given an object X, the operation X.lock() allows the invoking process to obtain exclusive access to X while X.unlock() allows it to release the lock. It is assumed that the locks are fair.

Let us observe that, as locks can be built from atomic registers in $\mathcal{ASM}_{n,0}[\emptyset]$ [38], both models $\mathcal{ASM}_{n,0}[\emptyset]$ and $\mathcal{ASM}_{n,0}$[lock] have the same computational power. This is no longer the case as soon as (even only) one process may crash ($t \geq 1$) [16].

System model $\mathcal{ASM}_{n,n-1}[\emptyset]$ This is the pure asynchronous wait-free computation model. Processes can communicate only through read/write registers and up to $t = n - 1$ processes may crash.

System model $\mathcal{ASM}_{n,n-1}$[Compare&Swap] This computation model is the wait-free $\mathcal{ASM}_{n,n-1}[\emptyset]$ model enriched with the Compare&Swap operation. This operation, denoted C&S(a, b), is on an atomic register say X. It does the following atomically: if the current value of X is a, it assigns b to X and returns *true*; otherwise, it returns *false*.

primitive X.C&S(old, new):
  if $(X = old)$ **then** $X \leftarrow new$; return$(true)$ **else** return$(false)$ **end if.**

This base operation exists on some machines such as Motorola 680x0, IBM 370, and on some SPARC architectures. In some cases, the returned value is not a boolean, but the previous value of X.

System model $\mathcal{ASM}_{n,n-1}[\diamond P]$ This is the wait-free computation model enriched with a failure detector of the class $\diamond P$. A failure detector is a device that provides processes with information on failures [7, 41]. According to the type and the quality of this information, several failure detector classes can be defined.

$\diamond P$ is the class of *eventually perfect* failure detectors [7]. A failure detector of this class provides each process p_i with a read-only set register, denoted $suspected_i$, that satisfies the following properties.

- Completeness. Eventually, the set $suspected_i$ of every correct process p_i contains the index of every faulty process.
- Eventual strong accuracy. Eventually, the set $suspected_i$ of every correct process p_i does not contain indexes of correct processes.

Assumptions that allow failure detectors of the class $\diamond P$ to be implemented and corresponding algorithms are described in [43].

Wait-free system model vs liveness property of an algorithm As we have seen previously the pure *wait-free* asynchronous read/write system model, namely, $\mathcal{ASM}_{n,nn-1}[\emptyset]$, considers that up to $t = n - 1$ processes may crash in a run.

The algorithms implementing an object in that model can be obstruction-free, non-blocking or wait-free. This means that, while an algorithm designed for $\mathcal{ASM}_{n,nn-1}[\emptyset]$ has to ensure the liveness property despite up to $n - 1$ process crashes, this liveness property it not necessarily wait-freedom (it can be obstruction-freedom or non-blocking).

3 A concurrent set object

This section presents algorithms that implements a concurrent set in the system model $\mathcal{ASM}_{n,0}$[lock] (i.e., the reliable asynchronous read/write shared memory model enriched with locks). Interestingly, the algorithms that are described still work when processes crash provided they do not crash while they own a lock.

The aim is here efficiency. The algorithms have to use locks as sparingly as possible and use them on small data.

3.1 Definition and assumptions

Definition A concurrent set object provides the processes with three operations.

- add(v) adds element v into the set. It returns *true* if v was not already present in the set. Otherwise it returns *false*.
- remove(v) suppresses element v from the set. It returns *true* if v was present in the set. Otherwise it returns *false*.
- contain(v) returns *true* if v is present in the set and *false* otherwise.

As already indicated any concurrent execution of this object has to be linearizable. It has to appear as if the operations have been invoked one after the other, and the corresponding sequence has to belong to specification of the set.

Assumptions It is assumed that the elements that the set can contain belong to a well-founded set. This means that they can be compared, have a smallest element, a greatest element and that there is a finite number of elements between any two elements.

As already indicated, it is assumed that the number of invocations of contain() is much bigger than the number of invocations of add() and remove(). This application-related assumption is usually satisfied when the set represents dictionary-like shared data structures.

3.2 The algorithm

As already said, the algorithm presented here is due to Heller, Herlihy, Luchangco, Moir, Scherer and Shavit [15]. The reader will also find in that reference a performance study based on experimental evaluation.

Design principles In order to be efficient the algorithm implementing the operation contain() has to be lock-free: a process has to terminate it whatever the concurrency pattern. Moreover, the algorithm implementing the operations add() plus remove() has to use locks "as little as possible" which means that locks have to be used parsimoniously.

The underlying data structures The set is represented by a linked list pointed to by a pointer kept in an atomic register *HEAD*. A cell of the list (say *NEW_CELL*) is made up of four atomic registers.

- *NEW_CELL.val* that contains a value (element of the set).
- *NEW_CELL.out* is a boolean (initialized to *false*) that is set to *true* when the corresponding element is suppressed from the list.
- *NEW_CELL.lock* is a lock used to ensure mutual exclusion on the registers composing the cell. This lock is accessed with the base operation lock() and unlock() (which can be easily implemented with underlying primitives such as $Test\&Set()/Reset()$).

- *NEW_CELL.next* is a pointer to the next cell. The set in organized as a sorted linked list. Initially the list is empty and contains two *sentinel* cells, as indicated in Figure 1. The values associated with these cells are the default values denoted $\bot$ and $\top$. These values cannot belong to the set and are such that for any value v of the set we have $\bot < v < \top$. All operations are based on a list traversal.

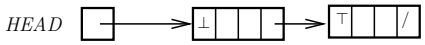

Figure 1: The initial state of the list

The remove(v) operation The algorithm implementing this operation is described at Lines 01-09 of Figure 4. Using the fact that the list is sorted in increasing order, the invoking process p_i traverses the list from the beginning until the first cell whose element v' is greater than v (lines 01-02). Then it locks the cell containing the element v' (that is pointed to by its local variable *curr*) and the immediately preceding cell (that is pointed to by its local variable *pred*).

The list traversal and the locking of the two consecutive cells are asynchronous and other processes can concurrently access the list to add or remove elements. It is consequently possible that there are synchronization conflicts that make the content of *pred* and *curr* no longer valid. More specifically, the cell pointed to by *pred* or *curr* could have been removed, or new cells could have been inserted between the cells pointed to by *pred* and *curr*. Hence, before suppressing the cell containing v (if any), p_i checks that *pred* and *curr* are still valid. The boolean procedure validate($pred, curr$) is used to that end (lines 10-11).

If the validation predicate is false, p_i restarts the removal operation (line 09). This is the price that to be paid to have an optimistic removal operation (there is no global locking of the whole list that would prevent concurrent processes from traversing the list). Let us remember that, as by assumption there are few invocations of the remove() and add() operations, p_i will eventually terminate its invocation.

If the validation predicate is satisfied, p_i checks whether v belongs to the set or not (boolean *pres*, line 05). If v is present, it is suppressed from the set (line 06). This is done in two steps.

- First the boolean field *out* of the cell containing v is set to *false*. This is a *logical* removal (logical, because the pointers have not yet been modified to suppress th cell from the list). This logical removal is denoted $S1$ in Figure 2.

- Then, the *physical* removal occurs. The pointer $(pred\uparrow).next$ is updated to its new value, namely $(curr\uparrow).next$. This physical removal is denoted $S2$ in Figure 2.

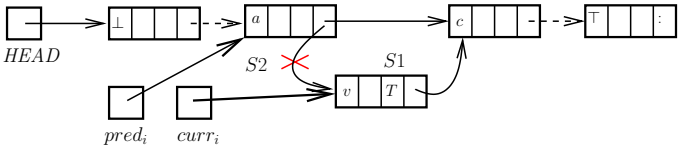

Figure 2: The remove() operation

The add(v) operation The algorithm implementing the add(v) operation is described at lines 12-23 of Figure 4. It is very close to the algorithm implementing the remove(v) operation. Process p_i first traverses the list until it reaches the cell whose value field is greater than v (lines 12-13) and then locks the cell that precedes it (line 14). Then, as previously, it checks if the values of its pointers *pred* and *curr* are valid (line 14). If they are and v is not in the list, p_i creates a new cell that contains v and insert it in the list (lines 17-20).

It is interesting to observe that, as in the removal operation, the addition of a new element v is done in two steps. The field $NEW\_CELL.next$ is first updated (line 18). This is the *logical* addition (denoted $S1$ in Figure 3). Only then, the field $(pred\downarrow).next$ is updated to a value pointing to NEW_CELL (line 18). This is the *physical* addition (denoted $S1$ in Figure 3).

Finally, p_i releases the lock on the cell pointed to by its local pointer variable ptr. It returns a boolean value if the validation predicate was satisfied and restarts if it was not.

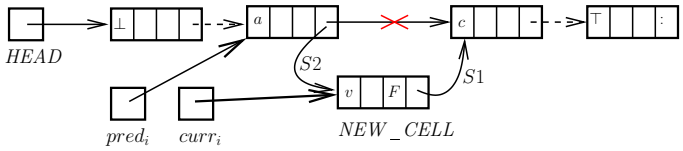

Figure 3: The add() operation

```
operation remove(v):        % (code for p_i) %
(01)  pred ← HEAD; curr ← (HEAD ↓).next;
(02)  while ((curr ↓).val < v) do pred ← curr; curr ← (curr ↓).next end while;
(03)  lock((pred ↓).lock); lock((curr ↓).lock); valid ← false;
(04)  if validate(pred, curr)
(05)     then valid ← true; pres ← ((curr ↓).val = v);
(06)          if (pres) then (curr ↓).out ← true; (pred ↓).next ← (curr ↓).next end if
(07)  end if;
(08)  unlock((pred ↓).lock); unlock((curr ↓).lock);
(09)  if (valid) then return(pres) else restart the operation end if.
=========================================================================
predicate validate(pred, curr):        % (code for p_i) %
(10) let res = (¬((pred ↓).out) ∧ ¬((curr ↓).out) ∧ ((pred ↓).next = curr));
(11) return(res).
=========================================================================
operation add(v):        % (code for p_i) %
(12)  pred ← HEAD; curr ← (HEAD ↓).next;
(13)  while ((curr ↓).val < v) do pred ← curr; curr ← (curr ↓).next end while;
(14)  lock((pred ↓).lock); valid ← false;
(15)  if validate(pred, curr)
(16)     then valid ← true; to_add ← (curr ↓).val ≠ v);
(17)          if (to_add) then NEW_CELL ← new_cell(); NEW_CELL.out ← false;
(18)                           NEW_CELL.val ← v; NEW_CELL.next ← curr;
(19)                           NEW_CELL.lock ← open; (pred ↓).next ← (↑ new_cell)
(20)          end if
(21)  end if;
(22)  unlock((pred ↓).lock);
(23)  if (valid) then return(to_add) else restart the operation end if.
=========================================================================
operation contain(v):        % (code for p_i) %
(24)  curr ← HEAD;
(25)  while ((curr ↓).val < v) do curr ← (curr ↓).next end while;
(26)  let res = (curr ↓).val = v) ∧ (¬(curr ↓).out);
(27) return(res).
```

Figure 4: Implementation of a concurrent set object in $\mathcal{ASM}_{n,0}$[lock] [15]

The contain(v) operation This operation is lock-free: it does not use locks and cannot be delayed by locks used by the add() and remove() operations. It consists of a simple traversal of the list. Let us remark that, during this traversal, the list does not necessarily remain constant: cells can be added or removed and then

the values of the pointers are not necessarily up to date when they are read by the process p_i that executes the contain operation. Let us consider Figure 5. It is possible that the pointer values $pred_i$ and $curr_i$ of the current invocation of contain(v) by p_i are as indicated on the figure while all the cells between the ones containing a_1 and b are removed and a new cell containing b is concurrently added.

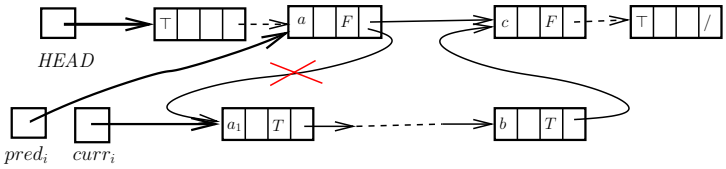

Figure 5: The contain() operation

The list traversal is the same as for the add() and remove() operations. The value *true* is returned if and only if v is currently the value of the cell pointed at by *curr* and this cell has not been logically removed. The algorithm relies on the fact that a cell cannot be recycled as long as it is reachable from a global or local pointer. (In contrast, cells that are no longer accessible can be recycled.)

3.3 Properties of the construction

Base properties The previous implementation of a concurrent set has the following noteworthy properties.

- The contain() operation is lock-free.
- The traversal of the list by an add()/remove() operation is lock-free (a cell locked by an add()/remove() does not prevent another add()/remove() from progressing until it locks a cell).
- Locks are used on at most two (consecutive) cells by an add()/remove() operation.
- There is no atomically markable reference (indicating that a cell is no longer meaningful requires to use a pointer to update the appropriate field of the pointed cell).
- Invocations of the add()/remove() operations on non-adjacent list entries do not interfere, thereby favoring concurrency.

Linearization points As already indicated, the linearization point of an operation invocation is a point of the time line such that the operation appears as if it has been executed instantaneously at that time instant. This point must lie between the starting time and the ending time of the operation.

The algorithm described in Figure 4 provide the operations add(), remove() and contain() with the following linearization points. Let an operation be *successful* (*unsuccessful*) if it returns *true* (*false*).

- remove() operation.
  - The linearization point of a successful remove(v) operation is when it marks the value v as being removed from the set, i.e., when it executes the statement $(curr\downarrow).out \leftarrow true$ (line 06).
  - The linearization point of an unsuccessful remove(v) operation is when, during its list traversal, it reads the first unmarked cell with a value $v' > v$ (line 02).
- add(v) operation.
  - The linearization point of a successful add(v) operation is when it updates the pointer $(pred\downarrow).next$ that, from then on, points to the new cell (line 19).
  - The linearization point of an unsuccessful add(v) operation is when it reads the value kept in $(curr\downarrow).val$ and that value is not v (line 16).

- contain(v) operation.
  - The linearization point of a successful contain(v) operation is when it checks whether the value v kept in $(curr\uparrow).val$ belongs to the set, i.e., $(curr\uparrow).out$ is then false (line 26).
  - The linearization point of an unsuccessful contain(v) operation is more tricky to define. This is due to the fact that (as discussed previously with the help of Figure 5), while contain(v) executes, an execution of add(v) or remove(v) can proceed concurrently.

    Let τ_1 be the time at which a cell containing v is found but its field *out* is marked *true* (line 26), or a cell containing $v' > v$ is found (line 25). Let τ_2 be the any time before the linearization point of a new operation add(v) that adds v to the set (if there is no such add(v), let $\tau_2 = +\infty$). The linearization point of an unsuccessful contain(v) operation is $\min(\tau_1, \tau_2)$.

This implementation of a concurrent set has been formally proved correct in [8].

4 A lock-free non-blocking queue

This section describes a non-blocking implementation of a queue due to Michael and Scott [34]. Interestingly, this implementation is included in the standard Java Concurrency Package. As we have seen, *non-blocking* means that, whatever the concurrency pattern, at least one process has to terminate its enqueue() or dequeue() operation.

The implementation of the queue does not rely on locks, it assumes an underlying Compare&Swap primitive, hence it is for the system model denoted $\mathcal{ASM}_{n,n-1}$[Compare&Swap], (Let us notice that Compare&Swap can be replaced by the pair of LL/SC primitives or the memory-to-memory Swap operation supplied by some machines. This is because they have the same computational power [16].)

The interested reader will find other lock-free algorithms for shared queues in [17, 31, 36, 50] and non-blocking algorithms for shared stacks and queues in [44] and for shared queues in [49].

4.1 Underlying data structure

The queue is implemented by a linked list as described in Figure 6. The core of the implementation consists then in handling pointers with the help of the Compare&Swap primitive.

The underlying list The list is accessed from an atomic register Q that contains a pointer to a record made up of two fields denoted *head* and *tail*. Each of these field is an atomic register.

Each atomic register $(Q\uparrow).head$ and $(Q\uparrow).tail$ has two fields denoted *ptr* and *tag*. The field *ptr* contains a pointer, while the field *tag* contains an integer (see below). To simplify the exposition, it is assumed that each field *ptr* and *tag* can be read independently.

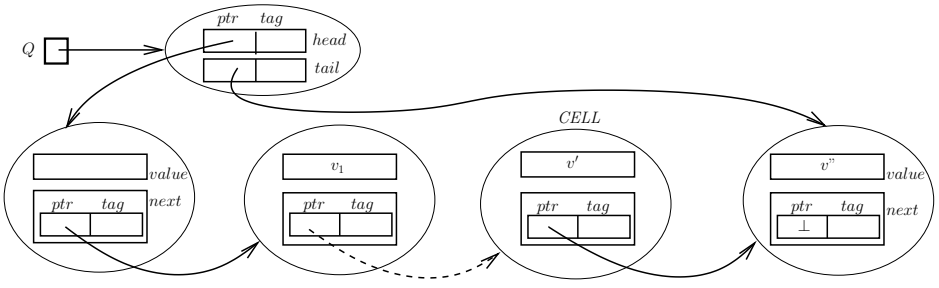

Figure 6: The list implementing the queue

The list is made up of cells such that the first cell is pointed to by $(Q\uparrow).head.ptr$ and the last cell of the list is pointed to by $(Q\uparrow).tail.ptr$.

Let *CELL* be a cell. It is a record composed of two atomic registers. The atomic register *CELL.value* contains a value enqueued by a process, while (similarly to $(Q\downarrow).head$ and $(Q\downarrow).tail$) the atomic register *CELL.next* is made up of two fields: *CELL.next.ptr* is a pointer to the next cell of the list (or $\perp$ if *CELL* is the last cell of the list) and *CELL.next.tag* is an integer.

Initially the queue contains no element but the list Q contains a dummy cell *CELL* (see Figure 7). This cell is such that *CELL.next.ptr* is (always) irrelevant and $CELL.next.ptr = \perp$. This dummy cell allows for a simpler algorithm. It always belongs to the list and $(Q\downarrow).head.ptr$ always points to it. Differently, $(Q\downarrow).tail.ptr$ points to the dummy cell only when the list is empty. Moreover, we have initially $(Q\downarrow).head.tag = (Q\downarrow).tail.tag = 0$.

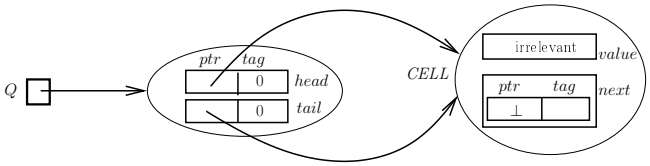

Figure 7: Initial state of the list

Compare&Swap primitive and the ABA issue The algorithms implementing the operations enqueue() and dequeue() consist basically in pointer management. To enqueue a new element a process prepares a new cell and has then to appropriately update pointers. Similarly, a dequeue consists in handling pointers. To that end, the algorithms use the hardware-provided Compare&Swap primitive.

When using Compare&Swap, a process p_i usually does the following. It first reads the atomic register X (obtaining value a), and later updates X to a new value c only if X has not been modified by another process since it has been read by p_i. Hence, p_i invokes $X.\mathsf{C\&S}(a, c)$. Unfortunately, the fact that this invocation returns *true* to p_i does not allow p_i to conclude that X has not been modified since the last time it read it. This is because between the read of X and the invocation $X.\mathsf{C\&S}(a, c)$ both issued by p_i, X could have been updated twice, first by a process p_j that has successfully invoked $X.\mathsf{C\&S}(a, b)$ and then by a process p_k that has successfully invoked $X.\mathsf{C\&S}(b, a)$, restoring thereby the value a into X. This is called the ABA problem.

This problem can be solved by associating tags (sequence numbers) with each value that is written. The atomic register X is then composed of two fields $\langle content, tag\rangle$. When it reads X, a process p_i obtains a pair $\langle x, y\rangle$ (where x is the current "data value" of X) and it later invokes $X.\mathsf{C\&S}(\langle x, y\rangle, \langle c, y+1\rangle)$ to write a new value c into X. It is easy to see that the write succeeds only if X has continuously been equal to $\langle x, y\rangle$. (As we are about to see in the enqueue() and dequeue() algorithms implementing the queue, the field *content* denotes a pointer value).

4.2 The dequeue(v) and enqueue() algorithms

As already indicated, these algorithms consist in handling pointers in an appropriate way. An interesting point is the fact that these algorithms require processes to help other process terminate their operations. Actually, this helping mechanism is the mechanism that implements the non-blocking property.

The enqueue() algorithm The algorithm implementing the enqueue() operation is described at lines lines 01-13 of Figure 8. The invoking process p_i first creates a new cell in the shared memory, assigns its address to the local pointer $\ell cell$ and updates its fields *value* and *next.ptr* (line 01). Then p_i enters a loop that it will exit when the value v has been enqueued.

In the loop, p_i executes the following statements. It is important to notice that, in order to obtain consistent pointer values, these statements include sequences of read and re-read (with Compare&Swap) to check that pointer values have not been modified.

```
operation enqueue(v):      % (code for p_i) %
(01)  ℓcell ←↑ new_cell(); (ℓcell ↓).value ← v; (ℓcell ↓).next.ptr ← ⊥;
(02)  repeat forever
(03)  ℓtail ← (Q ↓).tail;
(04)  ℓnext ← (ℓtail.ptr ↓).next;
(05)  if (ℓtail = (Q ↓).tail) then
(06)    if (ℓnext.ptr = ⊥)
(07)      then if ((ℓtail.ptr ↓).next).C&S(ℓnext, ⟨ℓcell, ℓnext.tag + 1⟩)
(08)             then (Q ↓).tail).C&S(ℓtail, ⟨ℓcell, ℓtail.tag + 1⟩); return(ok)
(09)           end if
(10)      else ((Q ↓).tail).C&S(ℓtail, ⟨ℓnext.ptr, ℓtail.tag + 1⟩)
(11)    end if
(12)  end if
(13)  end repeat.
==========================================================================
operation dequeue():      % (code for p_i) %
(14)  repeat forever
(15)  ℓhead ← (Q ↓).head;
(16)  ℓtail ← (Q ↓).tail;
(17)  ℓnext ← (ℓhead.ptr ↓).next;
(18)  if (ℓhead = (Q ↓).head) then
(19)    if (ℓhead.ptr = ℓtail.ptr)
(20)      then if (ℓnext.ptr = ⊥) then return(empty) end if;
(21)           ((Q ↓).tail).C&S(ℓtail, ⟨ℓnext.ptr, ℓtail.tag + 1⟩)
(22)      else result ← (ℓnext.ptr ↓).value;
(23)           if ((Q ↓).head).C&S(ℓhead, ⟨ℓnext.ptr, ℓhead.tag + 1⟩)
(24)             then free(ℓhead.ptr); return(result)
(25)           end if
(26)    end if
(27)  end if
(28)  end repeat.
```

Figure 8: Non-blocking implementation of a concurrent queue in $\mathcal{ASM}_{n,n-1}$[Compare&Swap] [34]

- Process p_i first makes local copies (kept in $\ell tail$ and $\ell next$) of $(Q\downarrow).tail$ and $(\ell tail.ptr\downarrow).next$, respectively. These values inform p_i on the current state of the tail of the queue (lines 03-04).
- Then p_i checks if the content of $(Q\downarrow).tail$ has changed since it read it (line 05). If it has changed, $\ell tail.ptr$ no longer points to the last element of the queue. Consequently, p_i starts the loop again.
- If $\ell tail = (Q\downarrow).tail$ (line 06), p_i optimistically considers that no other process is currently trying to enqueue a value. It then checks if $\ell next.ptr$ is equal to $\perp$.
  - If $\ell next.ptr = \perp$, p_i optimistically considers that $\ell tail$ points to the last element of the queue. It consequently tries to add the new element v to the list (lines 07-08) This is done in two steps, each based on a Compare&Swap: the first to append the cell to the list, the second to update the pointer $(Q\downarrow).tail$.
    - Process p_i tries first to append its new cell to the list. This is done by executing $((\ell tail.ptr\downarrow).next).\mathsf{C\&S}(\ell next, \langle \ell cell, \ell next.tag + 1\rangle)$ (line 07). If p_i does not succeed, this is because another process succeeded in appending a new cell to the list. If it is the case, p_i continues looping.
    - If process p_i succeeds in appending its new cell to the list, tries to update the content of $(Q\downarrow).tail$. This is done by executing $(Q\downarrow).tail).\mathsf{C\&S}(\ell tail, \langle \ell cell, \ell tail.tag + 1\rangle)$ (line 08). Finally, p_i returns from its invocation.

Let us observe that it is possible that the second Compare&Swap does not succeed. This is the case when, due to asynchrony, another process p_j did the work for p_i by executing line (line 10 of *enqueue*() or line 21 of *dequeue*().

- If $\ell next.ptr \neq \perp$, p_i discovers that $\ell next$ does not point to the last element of the queue. Hence, p_i discovers that the value of $(Q\downarrow).tail$ was not up to date when it read it. Another process has added an element to the queue but had not yet updated $(Q\downarrow).tail$ when p_i read it.

  In that case, p_i tries help the other process terminate the update of $(Q\downarrow).tail$ if not yet done. To that end, it executes $((Q\downarrow).tail).\mathsf{C\&S}(\ell tail, \langle \ell next.ptr, \ell tail.tag + 1\rangle)$ (line 10) before restarting the loop.

Linearization point of an enqueue() operation The linearization point associated with an enqueue() operation corresponds to the execution of the Compare&Swap statement of line 07. This means that an enqueue() operation appears as if it has been executed atomically when the new cell is linked to the last cell of the list.

The dequeue() algorithm The algorithm implementing the dequeue() operation is described at lines 14-28 of Figure 8. The invoking process loops until it returns a value at line 24. Due to its strong similarity with the algorithm implementing the enqueue() operation, the dequeue() algorithm is not described in details.

Let us notice that if $\ell head \neq (Q\downarrow).head$ (i.e., the predicate at line 18 is false), the head of the list has been modified while p_i is trying to dequeue an element. In that case, p_i restarts the loop.

If $\ell head = (Q\downarrow).head$ (line 18) then the values kept in $\ell head$ and $\ell next$ defining the head of the list are consistent. Process p_i then checks if $\ell head.ptr = \ell tail.ptr$, i.e., if (according to the values it has read at lines 15-16) the list consists currently in a single cell (line 19). If it is the case and this cell is the dummy cell (as witnessed by the predicate $\ell next.ptr = \perp$), the value *empty* is returned (line 20). In contrast, if $\ell next.ptr \neq \perp$, a process is concurrently adding a new cell to the list. To help it terminate its operation, p_i executes $((Q\downarrow).tail).\mathsf{C\&S}(\ell tail, \langle \ell next.ptr, \ell tail.tag + 1\rangle)$ (line 21).

Otherwise ($\ell head \neq (Q\downarrow).head$), there is at least one cell in addition to the dummy cell. This cell is pointed to by $\ell next.ptr$. The value kept in that cell can be returned (lines 22-24) if p_i succeeds in updating the atomic register $(Q\downarrow).head$ that defines the head of the list. This is done by $((Q\downarrow).head).\mathsf{C\&S}(\ell head, \langle \ell next.ptr, \ell head.tag + 1\rangle)$ (line 23). If this Compare&Swap succeeds, p_i returns the appropriate value and frees the cell (pointed to by $\ell next.ptr$ which has been suppressed from the list (line 24). Let us observe that the cell that is freed is the previous dummy cell while the cell containing the returned value v is the new dummy cell.

Linearization point of a dequeue() operation The linearization point associated with a dequeue() operation is the execution of the Compare&Swap statement of line 23 that terminates successfully. This means that an dequeue() operation appears as if it has been executed atomically when the the pointer to the head of the list $(Q\downarrow).head$ is modified.

Remark 1 Both linearization points correspond to the execution of successful Compare&Swap statements. The two other invocations of Compare&Swap statements (lines 10 and 23) constitute the helping mechanism that realize the non-blocking property.

Remark 2 It is important to notice that, due to the helping mechanism, the crash of a process does not annihilate the non-blocking property. If processes crash at any point while executing enqueue() or dequeue() operations, at least one process that does not crash while executing its operation terminates it.

Remark 3 The interested reader will find other object constructions based on similar principles in [1, 14, 35, 39, 40, 42].

5 Two wait-free objects

This section describes wait-free implementations of two objects, namely a splitter and a snapshot object, in the base read/write system, i.e., $\mathcal{ASM}_{n,n-1}[\emptyset]$. It is important to observe that this is the weakest shared memory model: processes can communicate by read/write registers only and up to $n-1$ processes may crash.

5.1 A wait-free splitter object

Definition A *splitter* is a concurrent object that provides processes with a single operation, denoted direction(). This operation returns a value to the invoking process. The semantics of a splitter is defined by the following properties [30, 37].

- Validity. The value returned by direction() is *right*, *down* or *stop*.
- Solo execution. If a single process invokes direction(), only *stop* can be returned.
- Concurrent execution. If x processes invoke direction(), then:
  - At most $x-1$ processes obtain the value *right*,
  - At most $x-1$ processes obtain the value *down*,
  - At most one process obtains the value *stop*.
- Termination. If a correct process invokes direction() it obtains a value.

The splitter object has been introduced in an implicit way by Lamport to implement fast mutual exclusion in failure-free systems [30]. It has then been captured "as an object" by Moir and Anderson [37] who used it to design a wait-free algorithm solving the renaming problem [3, 6].

A wait-free splitter algorithm The very elegant and simple algorithm described in Figure 9 implements a splitter [30]. The internal state of a splitter SP is made up of two atomic registers: $LAST$ that can contain a process index, and is initialized to any value, and a boolean $CLOSED$ initialized to *false*.

When a process p_i invokes SP.direction() it first writes its name in the atomic register $LAST$ (line 01). Then it checks if the "door" is open (line 02). If it has been closed by another process, p_i returns *right* (line 03). Otherwise, p_i closes the door (which can be closed by several processes, line 04) and then checks if it was the last process to invoke the direction() operation (line 05). If this is the case, p_i returns *stop*, otherwise it returns *down*.

```
operation SP.direction():      % (code for p_i) %
(01) LAST ← i;
(02) if (CLOSED)
(03)    then return(right)
(04)    else CLOSED ← true;
(05)         if (LAST = i)
(06)            then return(stop)
(07)            else return(down)
(08)         end if
(09) end if.
```

Figure 9: Wait-free implementation of a splitter object in $\mathcal{ASM}_{n,n-1}[\emptyset]$ [30, 37]

Remark A process that moves right is actually a *late* process: it arrived late at the splitter and found $CLOSED = true$. Differently, a process that moves down is actually a *slow* process: it set $LAST \leftarrow true$ but was not quick enough during the period that started when it updated $LAST$ (line 01) and ended when it read $LAST$ (line 05). At most one process can be neither late not slow, it is *on time* and gets *stop*.

5.2 A wait-free snapshot object

Definition A *multi-writer snapshot* object is made up of m atomic registers (components). It provides the processes with two operations denoted update() and snapshot().

- update(r, v) allows the invoking process to write a value v in component r ($1 \leq r \leq m$) of the snapshot object. The operation returns the control value ok.
- snapshot() allows the invoking process to obtain the values all components. It returns consequently an array of m values.

Let us remember that, as the implementation has to be linearizable, an invocation of snapshot() appears as if it has been executed instantaneously, hence it is as if the m values it returns have been read simultaneously.

The wait-free snapshot algorithm that follows is due to Imbs and Raynal [27].

Underlying shared data structures The wait-free snapshot algorithm that follows is due to Imbs and Raynal [27] (where a proof can be found). It uses two arrays of atomic registers.

- The first, denoted $REG[1..m]$, is made up of m atomic registers. Register $REG[r]$ is associated with component r. It has three fields $\langle value, pid, sn\rangle$ whose meaning is the following. $REG[r].value$ contains the current value v of the component r, while $REG[r].(pid, sn)$ is the "identity" of v. $REG[r].pid$ is the index of the process that issued the corresponding update(r, v) operation, while $REG[r].sn$ is the sequence number of this update when considering all updates issued by p_{pid}.
- The second array, denoted $HELPSNAP[1..n]$ is made up of n atomic registers, one per process. $HELPSNAP[i]$ is written only by p_i and contains a snapshot of $REG[1..m]$ computed by p_i during its last update() invocation. This snapshot value is destined to help processes that issued snapshot() invocations concurrent with p_i's update. More precisely, if during its invocation of snapshot() a process p_j discovers that it can be helped by p_i, it returns the value currently kept in $HELPSNAP[i]$ as output of its own snapshot invocation.

The local variable can_help_i is a set initialized to $\emptyset$ that will contain process identities.

```
operation snapshot():      % (code for p_i) %
(01) can_help_i ← ∅;
(02) for each r ∈ {1, ··· , m} do aa[r] ← REG[r] end for;
(03) while (true) do
(04)   for each r ∈ {1, ··· , m} do bb[r] ← REG[r] end for;
(05)   if (∀r ∈ {1, ··· , m} : aa[r] = bb[r]) then return(bb[1..m].value) end if;
(06)   for each r ∈ {1, ··· , m} such that bb[r] ≠ aa[r] do
(07)         let ⟨−, w, −⟩ = bb[r];
(08)         if (w ∈ can_help_i) then return(HELPSNAP[w])
(09)                             else  can_help_i ← can_help_i ∪ {w}
(10)         end if
(11)   end for;
(12)   aa ← bb
(13) end while.
=============================================================
operation update(r, v):    % (code for p_i) %
(14) sn_i ← sn_i + 1; REG[r] ← ⟨v, i, sn_i⟩;
(15) HELPSNAP[i] ← snapshot();
(16) return(ok).
```

Figure 10: Wait-free implementation of a snapshot object in $\mathcal{ASM}_{n,n-1}[\emptyset]$ [27]

The snapshot() operation The algorithm implementing the snapshot() operation (for process p_i) is described in Figure 10. The read of the array $HELPSNAP[1..n]$ (at line 02 or line 04) is called a *scan*. A scan is asynchronous and not atomic (the array is read in any order and at any speed, only the reading of each entry is atomic). As in [2], process p_i first uses a "sequential double scan" to try compute a snapshot value by itself. If it cannot terminate by itself, it looks for a process p_w which can help it. As we will see, this occurs when p_i observes that there is a process p_w that issued two update invocations while it (p_i) is still executing its snapshot invocation.

- *Try to terminate without help: successful double scan.*
  A process p_i first scans REG twice (line 02 and line 04). The important point here is that, when considering any two scans issued by a process, the second one always starts after the first one has terminated (scans issued by a process are sequential). The values obtained from the first scan are saved in the local array aa, while the values obtained from the second scan are saved in the local array bb.

  If the local predicate $\forall r : aa[r] = bb[r]$ is true, p_i has obtained the same values in both scans. This is called a *successful double scan*. This means that $REG[1..m]$ was containing these values at any time during the period starting at the end of the first scan and finishing at the beginning of the second scan. Consequently, p_i returns the array of values $bb[1..m].value$ as the result of its snapshot invocation (line 05).

- *Otherwise, try to benefit from the help of other processes.* If the predicate $\forall r : aa[r] = bb[r]$ is false, p_i looks for all entries r that have been modified during its previous double scan. Those are the entries r such that $aa[r] \neq bb[r]$. Let r be such an entry. As witnessed by $bb[r] = \langle -, w, - \rangle$, the component r has been modified by p_w.

  The predicate $w \in can\_help_i$ (line 08) is the helping predicate. It means that process p_w issued two updates that are concurrent with p_i's snapshot invocation. As we are about to see (Figure 11 and line 15 of the code of operation update(r, v) described in Figure 10), this means that p_w has issued a snapshot embedded in an update concurrent with p_i's snapshot invocation. If this predicate is true, the corresponding snapshot value, that has been saved in $HELPSNAP[w]$, can be returned by p_i as output of its snapshot invocation (line 08).

  If the predicate is false, process p_i adds the identity w to the set can_help_i (line 09). Hence, can_help_i (that is initialized to $\emptyset$, line 01) contains identities x indicating that process p_x has issued its last update while p_i is executing its snapshot operation. Process p_i then moves the array bb into the array aa (line 12) and enters the **while** loop again. As we can see, the lines 12 and 04 constitute a new double scan.

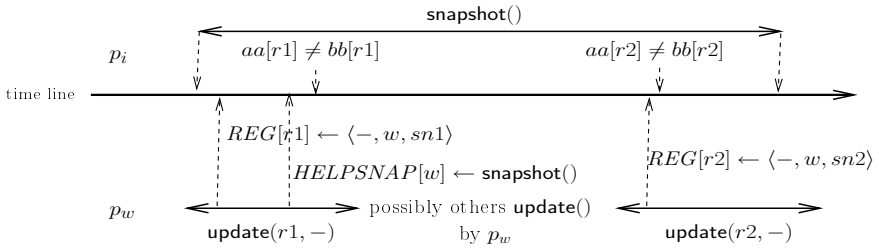

Figure 11: A snapshot with two concurrent updates by the same process

The update() operation The algorithm implementing the update(r, v) operation (for process p_i) is described in Figure 10. First, p_i increases the local sequence number generator sn_i (initialized to 0) and writes atomically the triple $\langle v, i, sn_i \rangle$ into $REG[r]$ (line 14). Then, p_i asynchronously computes a snapshot value (a size m array) and writes it into $HELPSNAP[i]$ (line 15). This constitutes the "write first, help later" strategy. The way $HELPSNAP[i]$ can be used by other processes as described previously. Finally, p_i returns from its update() invocation (line 16).

6 From obstruction-freedom to wait-freedom

This section presents first an obstruction-free implementation of a timestamping object that works in the pure wait-free asynchronous read/write shared memory model $\mathcal{ASM}_{n,n-1}[\emptyset]$ (Section 6.1). Then it shows how to enrich this system model in order to be able to transform any obstruction-free implementation into a wait-free implementation (Section 6.2).

6.1 Obstruction-free objects

As just indicated, this section considers the base unreliable read/write system model $\mathcal{ASM}_{n,n-1}[\emptyset]$.

The difficulty of obstruction-freedom Obstruction-freedom has been defined in the Introduction. It is a progress condition stating that if a process executes an operation in concurrency-free context it terminates its operation. It is important to see that several process may have started executing operations on the object. Hence, an obstruction-free implementation of a concurrent object must ensure two important things:

- The safety properties that define the object consistency have never to be violated (i.e., whatever the concurrency pattern on the operation invocations).
- Differently, the liveness property of each operation has to be guaranteed only if the invoking process does not crash during the operation and executes alone during a long enough period (namely, the period required to terminate the operation).

The difficulty in implementing obstruction-freedom lies in the fact that several processes may have pending operations. Even if after some time only one processes keeps on executing, other processes can have stopped at any statement of their operation execution.

A simple example Lots of algorithms implementing obstruction-free objects have been designed. One of the most popular is the double-ended queue presented in [18]. We present here a simpler concurrent object, namely, a timestamping object. Such an object X provides the processes with a single operation denoted get_timestamp() that returns an integer. No two invocations returns the same integer and, if invocation $\mathsf{gt}_1()$ returns before invocation $\mathsf{gt}_2()$ starts, the timestamp returned by $\mathsf{gt}_2()$ is greater than the one returned by $\mathsf{gt}_1()$.

It is easy to see that a lock-based implementation of a timestamp object is trivial: an atomic register protected by a lock is used to supply timestamps. But, as already indicated, locking and obstruction-freedom are incompatible in asynchronous crash-prone systems.

An algorithm The following obstruction-free implementation of a timestamp object has been proposed by Guerraoui, Kapalka and Kuznetsov in [11]. It relies on the underlying data structures.

- $NEXT$ defines the value of the next integer value that can be used as a timestamp.
- $LAST$ is an unbounded array of atomic registers. A process p_i deposits its index i in $LAST[k]$ to indicate it is trying to obtain the timestamp k.
- $COMP$ is another unbounded array of atomic boolean registers with each entry initialized to *false*. A process p_i sets $COMP[k]$ to *true* in order to indicate that it is competing for the timestamp k (hence several processes can write *true* into $COMP[k]$).

The algorithm implementing the obstruction-free operation get_timestamp() is described in Figure 12. It is inspired from the wait-free algorithm described in Figure 9 that implements a splitter. (The pair of registers $LAST[k]$ and $COMP[k]$ in Figure 12 plays the same role as the registers $LAST$ and $CLOSED$ in Figure 9.) A process p_i first reads the next possible timestamp value (register $NEXT$). Then it enters a loop that it will exit after it has obtained a timestamp (line 06).

```
operation get_timestamp(i):     % code for p_i %
(01)  k ← NEXT;
(02)  repeat forever
(03)     LAST[k] ← i;
(04)     if (¬COMP[k])
(05)        then COMP[k] ← true;
(06)             if (LAST[k] = i) then NEXT ← NEXT + 1; return(k) end if
(07)     end if;
(08)  k ← k + 1
(09)  end repeat.
```

Figure 12: Obstruction-free implementation of a timestamp object in $\mathcal{ASM}_{n,n-1}[\emptyset]$ [11]

In the loop, p_i first writes its index in $LAST[k]$ to indicate that it is the last process competing for the timestamp k (line 03). Then if it finds $COMP[k] = false$, p_i sets it to *true* to indicate that processes are competing for timestamp k. Let us observe that it is possible that several processes find $COMP[k]$ equal to *true* and set it to *true* (lines 04-05). Then, p_i checks the predicate $LAST[k] = i$. If this predicate is satisfied, p_i can conclude that it is the last process that wrote into $LAST[k]$. Consequently, all other processes (if any) competing for timestamp k will find $COMP[k]$ equal to *false*, and will directly proceed to line 08 to try to obtain timestamp $k + 1$. Hence, they don not execute lines lines 05-06.

It is easy to see that if, after some time, a single process keeps on executing its get_timestamp() invocation, it eventually obtain a timestamp. In contrast, when several processes find $COMP[k]$ equal to *false*, there is no guarantee that one of them obtains the timestamp k.

6.2 From obstruction-freedom to wait-freedom

An important question related to fault-tolerance is the following: How to boost an obstruction-free implementation of an object to a wait-free implementation? To answer this question, this section considers the enrichment of $\mathcal{ASM}_{n,n-1}[\emptyset]$ with a failure detector of the class $\diamond P$, i.e., the system model $\mathcal{ASM}_{n,n-1}[\diamond P]$.

Failure detector-based contention manager The answer to the previous question relies on the notion of a *contention manager*. Such an object CM can be used by an obstruction-free implementation in order to obtain a wait-free implementation. It provides each process p_i with two operations denoted contender(i) and finished(i). The former is used by p_i to indicate that it suspects that concurrency will prevent it from terminating, while p_i invokes the latter one to indicate that it does no longer compete to terminate its operation.

In our case, the algorithm described in Figure 12 is enriched as follows in order to obtain a wait-free implementation.

- When p_i executes line 08, it knows that there is concurrency to obtain timestamp k. Hence, the invocation CM.contender(i) is added to that line.
- When p_i executes return() (line 06), it is no longer competing to acquire a timestamp. Hence, before returning, it invokes CM.finished(i) to inform of it the contention manager.

The weakest failure detector to boost obstruction-freedom to wait-freedom The issue raised to solve the previous boosting has been moved to the implementation of a general contention manager object CM, and the previous question becomes: What is the weakest information on failures needed to implement such an object?

This question has been answered in [11] where it is shown that $\diamond P$ [7] is this weakest class of failure detectors that allows boosting obstruction-freedom to wait-freedom.

$\diamond P$-based contention manager A $\diamond P$-based contention manager is described in Figure 13. let us remember (see Section 2.2) that $\diamond P$ provides each process p_i with a set $suspected_i$ that eventually contains all faulty processes and only them.

The contention manager uses an underlying operation, denoted weak_ts(), that generates locally increasing timestamps and are such that if a process obtains a timestamp value ts, then any process can obtain timestamp values lower than ts a finite number of times only. This operation weak_ts() can be implemented from atomic read/write registers only. It is easy to see that weak_ts() is an operation weaker than get_timestamp().

```
operation contender(i):     % code for p_i %
(01)  if (TS[i] = 0) then TS[i] ← weak_ts() end if;
(02)  repeat compet_i ← {j | TS[j] ≠ 0) ∧ j ∉ suspected_i};
(03)           let ⟨ts, j⟩ be the smallest pair ∈ {⟨TS[x], x⟩ | x ∈ compet_i}
(04)  until (j = i) end repeat.
=============================================================
operation finished(i):     TS[i] ← 0.
```

Figure 13: $\diamond P$-based contention manager in $\mathcal{ASM}_{n,n-1}[\diamond P]$ [11]

The processes access an array of atomic registers TS that has one entry per process. This array is initialized to $[0, \ldots, 0]$. When p_i invokes contender(i), it assigns a weak timestamp to $TS[i]$ (line 01). It will reset $TS[i]$ to 0 when it executes finshed(i). Hence, $TS[i] \neq 0$ means that p_i is competing inside the contention manager. After it has assigned a value to $TS[i]$, p_i waits (loops) until the pair $(TS[i], i)$ is the smallest (according to lexicographical order) among the processes competing inside the contention manager that are not locally suspected to have crashed (lines 02-04). On can easily see that this contention management mechanism ensures that any process will eventually have priority to execute its object operation. A proof is given in [11].

7 Concluding remark (in one way or another anything is consensus!)

The consensus problem Consensus is one of the most important problems of fault-tolerant asynchronous computing. It is an *agreement* problem defined as follows. Every process is assumed to propose a value and (a) each non-faulty process has to decide a value (termination), such that (b) a decided value is a proposed value (validity) and (c) no two processes decide different values (agreement).

This agreement problem is central in a lot of coordination problems and is at the core of many fault-tolerant algorithms. In one way or another processes have to "agree" in nearly all applications. Unfortunately, the most important result associated with the consensus problem is the impossibility to solve it in $\mathcal{ASM}_{n,n-1}[\emptyset]$ (i.e., in the pure asynchronous wait-free read/write model) [10, 32].

The notion of consensus number Let $\mathcal{ASM}_{n,n-1}[X]$ be the system model $\mathcal{ASM}_{n,n-1}[\emptyset]$ enriched with (as many as we want) objects X. A fundamental question of concurrent programming in the presence of asynchrony and process crashes is the following: Given an object A (defined by a sequential specification), is there a wait-free implementation of A in $\mathcal{ASM}_{n,n-1}[X]$?

This question is answered by the consensus number notion [16]. The consensus number of a concurrent object X is the maximum number n of processes (or $+\infty$ if there is no such integer) for which one can solve the consensus problem in $\mathcal{ASM}_{n,n-1}[X]$ [16]. As an example, the consensus number of an atomic register is 1, the consensus number of Test&Set is 2. etc., and the consensus number of Compare&Swap (or the pair LL/SC) is $+\infty$.

Concurrent objects actually define an infinite hierarchy of objects (called *consensus hierarchy* or *wait-free hierarchy*) [16] such that the objects at level x are exactly those objects with consensus number x. Hence, atomic register is at level 1 while Compare&Swap $+\infty$ is at the highest level of the hierarchy.

This ranking of "synchronization objects" is a very powerful tool to understand the relative power of these objects. It can help decide which synchronization primitive a multi-core architecture must support.

The notion of a universal construction An algorithm that constructs a wait-free implementation of an object A in $\mathcal{ASM}_{n,n-1}[X]$ where the consensus number of X is $\geq n$, is called a *universal construction* for a size n system. Several universal constructions have been designed (e.g.,[12, 16]).

The reader interested in consensus number and universal constructions will find in depth developments on these notions in the following books [4, 19, 33, 43, 45].

On progress conditions Obstruction-freedom, non-blocking and wait-freedom are not the only progress conditions proposed so far. Asymmetric progress conditions have recently been defined [25, 26]. The definition and investigation of new progress conditions have received a new impetus and are becoming a very active research area. The interested reader can consult the very last results in [25, 26, 46, 47, 48].

References

[1] Afek Y., Weisberger E. and Weisman H., A Completeness Theorem for a Class of Synchronization Objects. *Proc. 12th Int'l ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 159-168, 1993.

[2] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.

[3] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524-548, 1990.

[4] Attiya H. and Welch J., Distributed Computing: Fundamentals, Simulations and Advanced Topics, (2d Edition), *Wiley-Interscience*, 414 pages, 2004.

[5] Bernstein Ph.A., Hadzilacos V. and Goodman N., Concurrency Control and Recovery in Database Systems. *Addison Wesley Publishing Company*, 370 pages, 1987.

[6] Castañeda A., Rajsbaum S. and Raynal M., The Renaming Problem in Shared Memory Systems: an Introduction. *Tech Report* #1960, IRISA, Université de Rennes 1 (France), November 2010 (Submitted to Journal Publication).

[7] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

[8] Colvin R., Groves L., Luchangco V. and Moir M., Formal Verification of a Lazy Concurrent List-based Set Algorithm. *Proc. 18th Int'l Conference on Computer Aided Verification (CAV'06)*, Springer Verlag LNCS #4144, pp. 475-488, 2006.

[9] Dijkstra E.W.D., Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(1):115-138, 1971.

[10] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[11] Guerraoui R., Kapalka M. and Kuznetsov P., The Weakest Failure Detectors to Boost Obstruction-freedom. *Distributed Computing*, 20(6): 415-433, 2008.

[12] Guerraoui R. and Raynal M., A Universal Construction for Wait-free Objects. *Proc. ARES 2007 Int'l Workshop on Foundations of Fault-tolerant Distributed Computing (FOFDC 2007)*, IEEE Press, pp. 959-966, 2007

[13] Guerraoui R. and Raynal M., From Unreliable Objects to Reliable Objects: the Case of Atomic Registers and Consensus. *9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, Springer Verlag LNCS #4671, pp. 47-61, 2007.

[14] Harris T.L., Fraser K. and Pratt I.A., A Practical Multi-word Compare-and-Swap Operation. *Proc. 16th Int'l Symposium on Distributed Computing (DISC'02)*, Springer Verlag LNCS #2508, pp. 265-279, 2002.

[15] Heller S., Herlihy M.P., Luchangco V., Moir M., Scherer W.III and Shavit N., A Lazy Concurrent List-Based Algorithm. *Parallel Processing Letters*, 17(4):411-424, 2007.

[16] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[17] Herlihy M.P., Luchangco V., Marin P. and Moir M., Non-blocking Memory Management Support for for Dynamic-size Data Structures. *ACM Transactions on Computers Systems*, 23(2):146-196, 2005.

[18] Herlihy M.P., Luchangco V. and Moir M., Obstruction-free Synchronization: Double-ended Queues as an Example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, pp. 522-529, 2003.

[19] Herlihy M.P. and Shavit N., Herlihy M.P. and Shavit N., The Art of Multiprocessor Programming, *Morgan Kaufman Pub.*, San Francisco (CA), 508 pages, 2008.

[20] Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[21] Hewitt C.E. and Atkinson R.R., Specification and Proof Techniques for Serializers. *IEEE Transactions on Software Engineering*, SE5(1):1-21, 1979.

[22] Hoare C.A.R., Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549-557, 1974.

[23] Horning J.J. and Randell B., Process Structuring. *ACM Computing Surveys*, 5(1):5-30, 1973.

[24] Imbs D. and Raynal M., A Note on Atomicity: Boosting Test&Set to Solve Consensus. *Information Processing Letters*, 109(12):589-591, 2009.

[25] Imbs D. and Raynal M., The x-Wait-freedom Progress Condition. (Distinguished paper). *Proc. 16th Int'l European Parallel Computing Conference (EUROPAR'10)*, Springer-Verlag LNCS #6271, pp. 584-595, 2010.

[26] Imbs D., Raynal M. and Taubenfeld G., On Asymmetric Progress Conditions. *29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 55-64, July 2010.

[27] Imbs D. and Raynal M., A Simple Snapshot Algorithm for Multicore Systems. *Proc. 5th IEEE Latin-American Symposium on Dependable Computing (LADC'11)*, IEEE Computer Press, Sao Paulo, March 2011.

[28] Lamport L., How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979.

[29] Lamport. L., On Interprocess Communication, Part 1: Models, Part 2: Algorithms. *Distributed Computing*, 1(2):77-101, 1986.

[30] Lamport L., A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1-11, 1987.

[31] Ladam-Mozes E. and Shavit N., An Optimistic Approach to Lock-free FIFO queues. *Proc. 18th Int'l Symposium on Distributed Computing (DISC'04)*, Springer Verlag LNCS #3274, pp. 117-131, 2004.

[32] Loui M. and Abu-Amara H., Memory Requirements for for agreement among Unreliable Asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press Inc., 1987.

[33] Lynch N.A., Distributed Algorithms. *Morgan Kaufman Pub.*, San Francisco (CA), 872 pages, 1996.

[34] Michael M.M. and Scott M.L., Simple, Fast and Practical Blocking and Non-Blocking Concurrent Queue Algorithms. *Proc. 15th Int'l ACM Symposium on Principles of Distributed Computing (PODC'96)*, pp. 267-275, 1996.

[35] Moir M., Practical Implementation of Non-Blocking Synchronization Primitives. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 219-228, 1997.

[36] Moir M., Nussbaum D., Shalev 0. and Shavit N., Using Elimination to to Implement Scalable and Lock-free FIFO Queues. *Proc. 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'05)*, ACM Press, pp. 253-262, 2005.

[37] Moir M. and Anderson J., Wait-Free Algorithms for Fast, Long-Lived Renaming. *Science of Computer Programming*, 25(1):1-39, 1995.

[38] Raynal M., Algorithms for Mutual Exclusion. *The MIT Press*, ISBN 0-262-18119-3, 107 pages, 1986.

[39] Raynal M., Synchronization is Coming Back, But is it the Same? (Keynote Speech). *IEEE 22nd Int'l Conference on Advanced Information Networking and Applications (AINA'08)*, pp. 1-10, Okinawa (Japan), 2008.

[40] Raynal M., Locks Considered Harmful: a Look at Non-traditional Synchronization. *Proc. 6th Int'l Workshop on Software Technologies for Future Embedded and Ubiquitous Computing Systems (SEUS'08)*, Springer Verlag LNCS #5287, pp. 369-380, 2008.

[41] Raynal M., Failure Detectors for Asynchronous Distributed Systems: an Introduction. *Wiley Encyclopdia of Computer Science and Engineering*, Vol. 2, pp. 1181-1191, 2009 (ISBN 978-0-471-38393-2).

[42] Raynal M., Shared Memory Synchronization in Presence of Failures: an Exercise-based Introduction. *IEEE Int'l Conference on Complex, Intelligent and Software Intensive Systems (CISIS'09)*, IEEE Computer Society Press, pp. 9-18, Fukuoka (Japan), 2009.

[43] Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. *Morgan & Claypool Publishers*, 251 pages, 2010 (ISBN 978-1-60845-293-4).

[44] Shafiei N., Non-blocking Array-based Algorithms for Stacks and Queues. *Proc. th Int'l Conference on Distributed Computing and Networking (ICDCV'09)*, Springer Verlag LNCS #5408, pp. 55-66, 2009.

[45] Taubenfeld G., Synchronization Algorithms and Concurrent Programming. *Pearson Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.

[46] Taubenfeld G., Contention-Sensitive Data Structure and Algorithms. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer Verlag LNCS #5805, pp. 157-171, 2009.

[47] Taubenfeld G., On the Computational Power of Shared Objects. *Proc. 13th Int'l Conference On Principle Of Distributed Systems (OPODIS 2009)*, Springer Verlag LNCS #5923, pp. 270-284, 2009.

[48] Taubenfeld G., The Computational Structure of Progress Conditions. *Proc. 24th Int'l Symposium on Distributed Computing (DISC'10)*, Springer Verlag LNCS #6343, pp. 221-235, 2010.

[49] Tsigas Ph. and Zhang Y., A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. *Proc. 13th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'01)*, ACM Press, pp. 134-143, 2001.

[50] Valois J.D., Implementing Lock-free Queues. *Proc. 7th Int'l Conference on Parallel and Distributed Computing Systems (PDCS'94)*, pp. 64-69, 1994.